365 DataScience

R PROGRAMMING for Statistics and Data Science

COURSE NOTES – SECTIONS 2,3,4,5

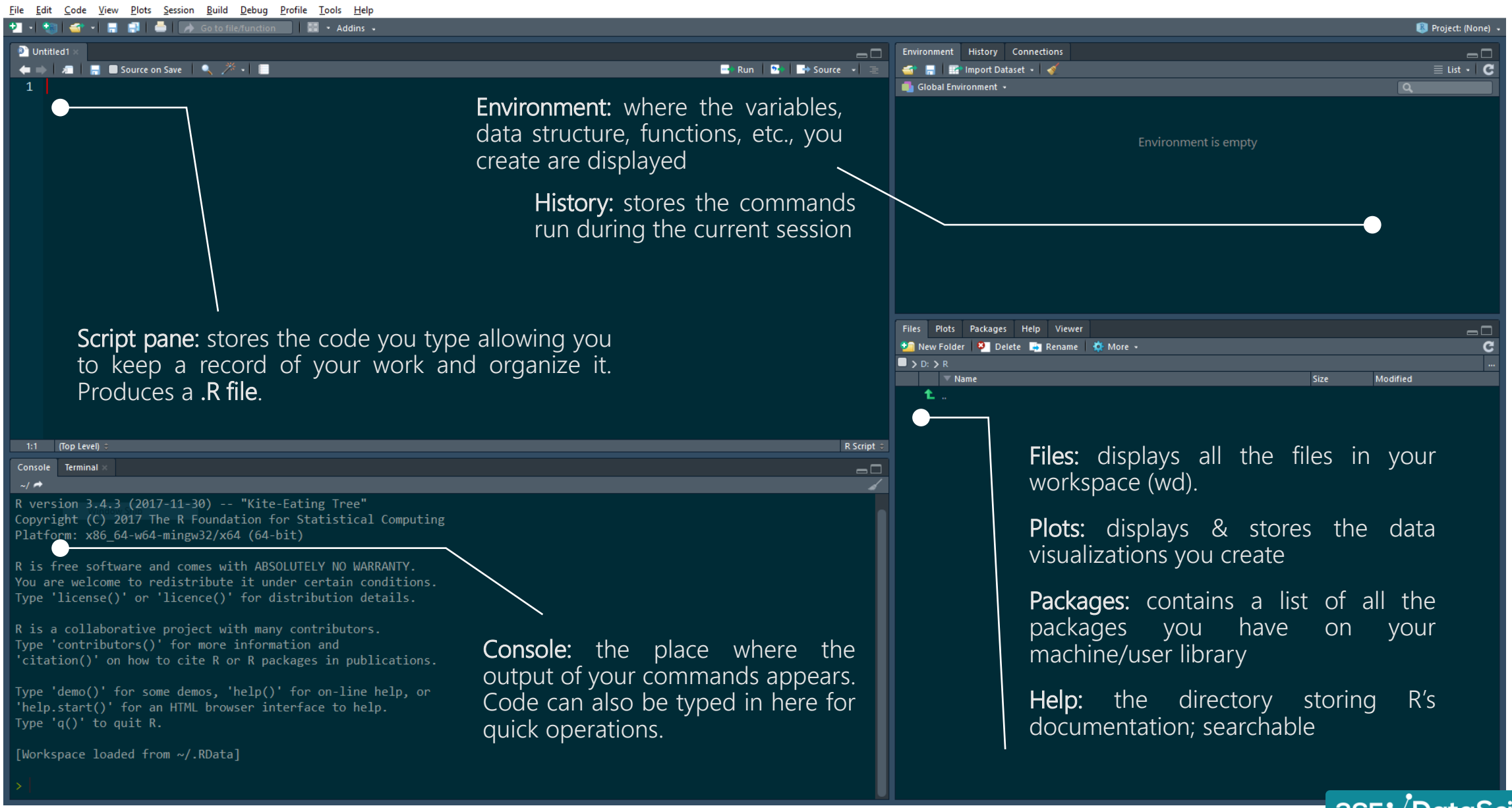
Words of welcome

You are here because you want to work better with data. Becoming comfortable with using a programming language for your data manipulations is a gigantic step forward. This way you will be able to automate, reproduce, and communicate your manipulations faster and better.

R is (perhaps) the best language to start with: it is an open-source language that is heavily tailored for handling data: from data wrangling through data visualization to machine learning. R's functionality is huge and, kind of like our universe, R's universe is always expanding.

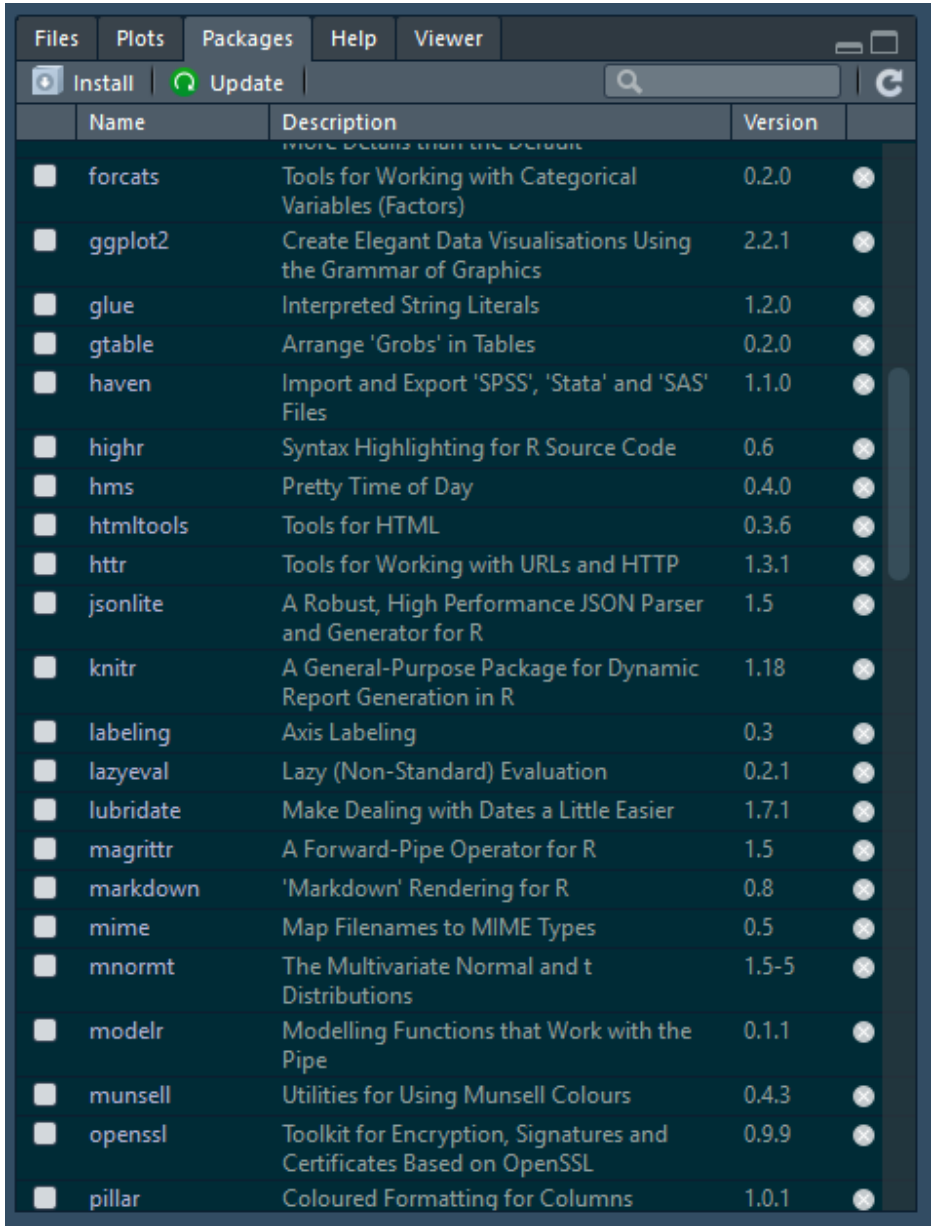
Packages are the main source of functionality in R: they comprise different families of functions intended for a specific manipulation or field. The majority of R's packages are stored on the CRAN project website https://cran.r-project.org/, but there are CRAN-external sources, too.

The RStudio interface


File Edit Code View Plots Session Build Debug Profile Tools Help
Addins
Project: (None)
Untitled1
Source on Save
Run
Source
1
Environment History Connections
Import Dataset
List
Global Environment
Environment is empty
Files Plots Packages Help Viewer
New Folder
Delete
Rename
More
D: > R
Name
Size
Modified
..
1:1
(Top Level)
R Script
Console
Terminal
~/
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
[Workspace loaded from ~/.RData]
>
Environment: where the variables, data structure, functions, etc., you create are displayed
History: stores the commands run during the current session
Script pane: stores the code you type allowing you to keep a record of your work and organize it. Produces a .R file.
Console: the place where the output of your commands appears. Code can also be typed in here for quick operations.
Files: displays all the files in your workspace (wd).
Plots: displays & stores the data visualizations you create
Packages: contains a list of all the packages you have on your machine/user library
Help: the directory storing R's documentation; searchable

365 DataScience

Packages



Preloaded packages: R has a decent functionality before installing new packages. However, R's {base} packages are often outdated, or cumbersome to use.

Installing packages: installing a package saves the package in our User Library for future use. A package needs to be installed only once.

Loading packages: If you want to use the functions in a specific package, you must tell R to load it for you. You only need to load a package once per session. You do that with the `library(package.name)` function.

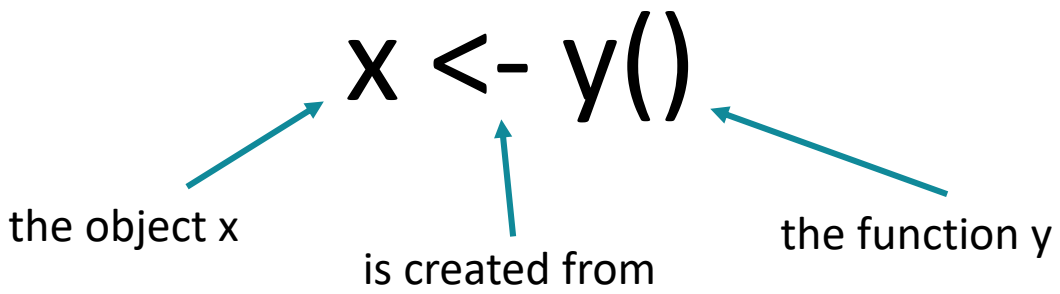
Distinguishing between functions: some packages have functions whose names overlap. You can recognise which package a function belongs to by checking the `{package.name}` next to it during autocompletion.

Removing packages: you can uninstall a package with the `remove.packages("package.name")` function.

365 DataScience

Creating an object

Objects are **named data structures** inside of which you store data. An object can be a single digit, a character, a Boolean value, a sentence, a data frame, a list of data frames, a multi-dimensional structure, and so on. You use functions to create objects.



The object name:

- Must begin with a small letter.
- Longer names can be created by separating the individual words with a dot (.), an underscore (_), or by capitalizing the first letter of every new word (objectName).
- Once you select a notation, stick with it

Once you store data inside an object, you can use the object name to call that data and do operations with it. An operation will be carried out on each element of the data structure, systematically.

365 DataScience

Data types

Integer

An *integer* is a **whole number**; any number that doesn't need anything after the decimal point is an integer.

R doesn't usually jump to creating an integer vector when you pass numbers. Often, you need to **explicitly tell** it to do so.

You do that by passing the letter **L** after each number in the integer object you're creating.

Double

Doubles store **regular numbers**: they can be large, small, positive, negative, with digits after the decimal or without.

Because R is heavily used for data analysis and most operations typically either involve or result in a double, this is **R's default way of saving numerical data**.

Character

Character vectors* store **text data**. A character element can be a single letter, number, or a symbol, or a longer string, like a sentence or even a paragraph.

To create a character string you must pass the value you want stored as a string in **quotation marks**.

Logical

Logical vectors store **Boolean data**; these are TRUE and FALSE values.

TRUE and FALSE and T and F can be used interchangeably as R recognises both. It is better, however, to use TRUE and FALSE because these are the reserved words which cannot be overwritten by the user.

TRUE and FALSE values must be inputted in **capital letters**, and without quotation marks.

`typeof(object.name)` returns the basic data type of the object you pass

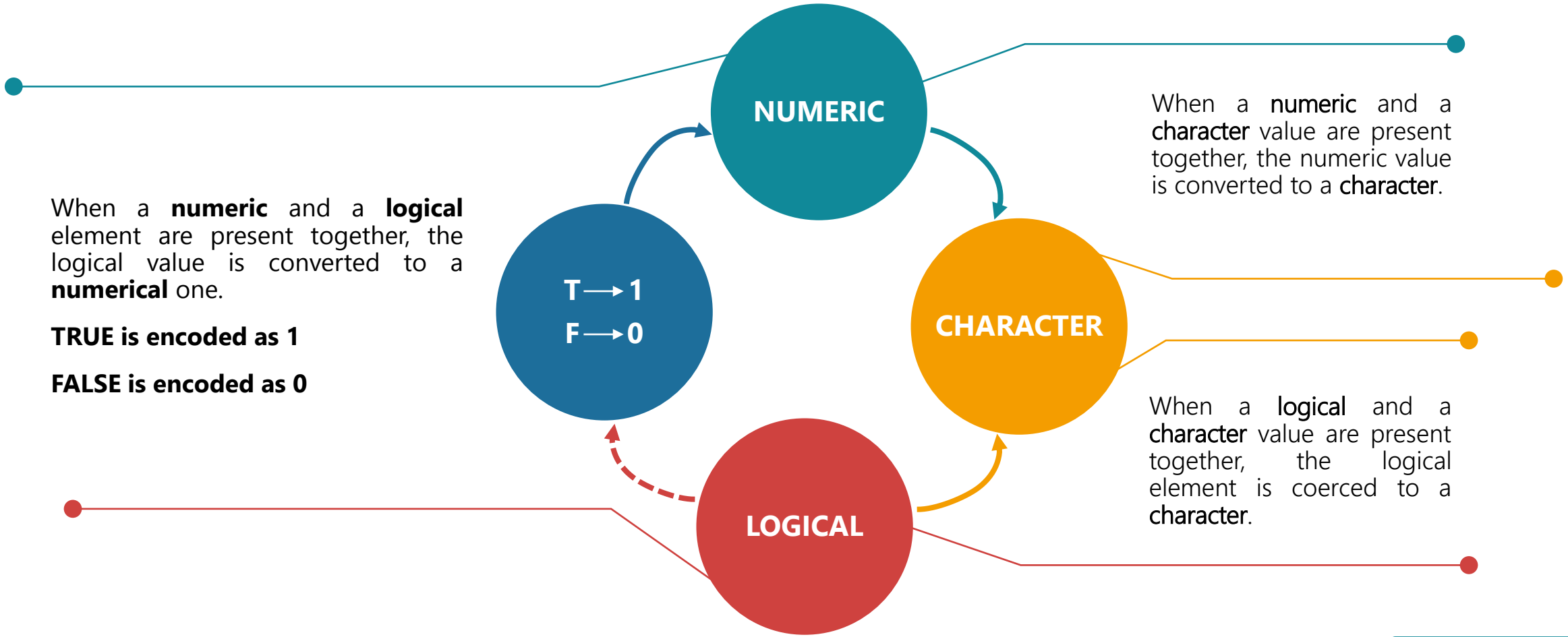
`is.integer(object.name)` returns a Boolean signifying whether the object you pass is an integer or not; an analogous command exists for the other data types, too

Vector - a sequence of data elements that are of the **same type**

365 DataScience

Coercion rules

R has ways to prevent certain mistakes from happening. For example, if you are trying to create an object, and you are passing the wrong type of data as an argument, then R will **convert** the value to the correct type, so you can end up creating your object. The *correct type* is typically the simplest type necessary to represent all the information.



When a **numeric** and a **logical** element are present together, the logical value is converted to a **numerical** one.

TRUE is encoded as 1

FALSE is encoded as 0

When a **numeric** and a **character** value are present together, the numeric value is converted to a **character**.

When a **logical** and a **character** value are present together, the logical element is coerced to a **character**.

365 DataScience

Functions and arguments

Think of a function as any other type of object in R. It has a name, stores information (a set of statements organized in a particular way so as to perform a specific task), and can be called on when needed.

```
function.name(argument1 = , argument2 = , … argumentN = NULL)
```

some arguments can have default values (e.g., argument = NULL)

```
function.name <- function(){
# body of code
return(value)
}
```

creates the function

the statements the function executes when called

what the function returns

Basics

- **Functions take arguments** – the arguments of a function can be data, additional instructions on how to carry out the operation, other functions (called **nesting**)
- **To run a function**, call the function and pass in parentheses the data you want the function to operate on; this is an argument
- **To save the result of a function** into an object for further use, use the object-creating formula (`object <- function(data)`)
- **The arguments of a function have an inherent order** when passing values you can either explicitly specify the argument they are for (and thus not follow the inherent order), or you can omit the argument name but keep to the inherent argument order.

Intermediate

- A function is comprised of three parts: the **function name**, the **body of code** (containing the statements), and **arguments**;
- Sometimes a fourth component can be included in the basic structure of a function – the **return value**; it specifies what the function returns once executed and is written last in the body

Vector operations

Vector – a sequence of data elements that are of the same type

Vector operations happen in an element-wise fashion

Vector recycling happens when if the two vectors you are doing operations with are different in length; in that case, R repeats the shorter vector until it matches the length of the longer one.

Example: operation with same-length vectors

v3	<-	v1	×	v2

Multiplying two vectors happens element-by-element

2	×	1	=	2
4		3		12
6		5		30
8		7		56
v1		v2		v3

Note that the first element of v1 is multiplied by the first element of v2, the second element of v1, by the second element of v2, and so on.

The resulting v3 is the **same length** as v1 and v2.

Example: vector recycling

v3	<-	v1	×	v2

R repeats the shorter vector (note that **the vector is not then saved this way**!)

2	×	1	=	2	×	1	=	2
4		3		4		3		12
		5		2		5		10
		7		4		7		28
v1		v2		v1		v2		v3

The resulting v3 is the **same length as the longer vector**, v2.

If the shorter vector is not a multiple of the longer one, R will issue a warning, but it will still carry out the operation.

365 DataScience

Vector attributes

Attributes are *additional information* about an object stored in the object; they do not affect the values of the object, instead they provide us with extra functionality if a function is designed to take into account whether or not an object has a specific attribute

`attributes(object.name)` checks whether the object has any attributes; an output `NULL` means the attributes object is empty

Names

You can give each element in a vector a name value.
When printed, R will display the values with their names above them.

```
George   John   Paul  Ringo
    23     26     24     26
> 
```

Assign names by creating a character vector and using the `names(object) <- c("name.1", "name.2" … "name.k")` command

Use the same command to **change the names** if needed

Remove names by setting the names values to null
`names(object) <- NULL`

Dimensions

Changing the dimensions of a vector enables you to transform your 1-dimensional vector into an as-many-dimensional object you want.

You can create a 2-dimensional matrix from a vector by specifying row and column values in the dim() attribute.

```
dim(object) <- c(3, 4)
```

The **row value comes first**, followed by the columns value.

Example:

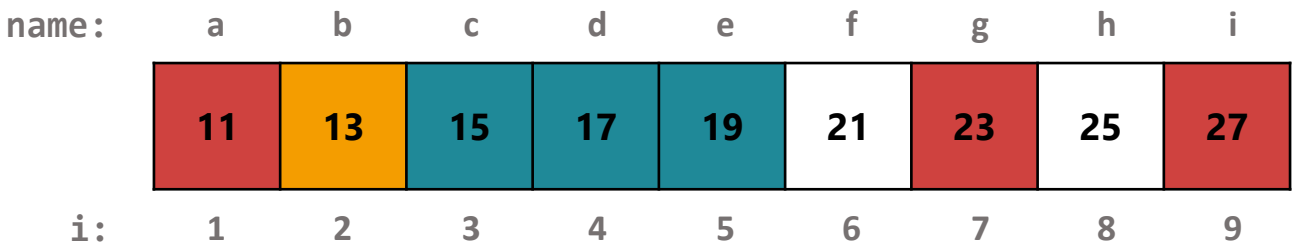
```
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
> dim(x) <- c(3, 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> 
```

Notice that R fills out the matrix **column by column**

365 DataScience

Vector slicing and recycling

- Indexing refers to selecting and extracting a single element from a structure (in this case, a vector)
- Slicing is to select and extract a sequence of elements
- R's notation for selecting a value is `object[i]`, where `i` is the index to which your value corresponds



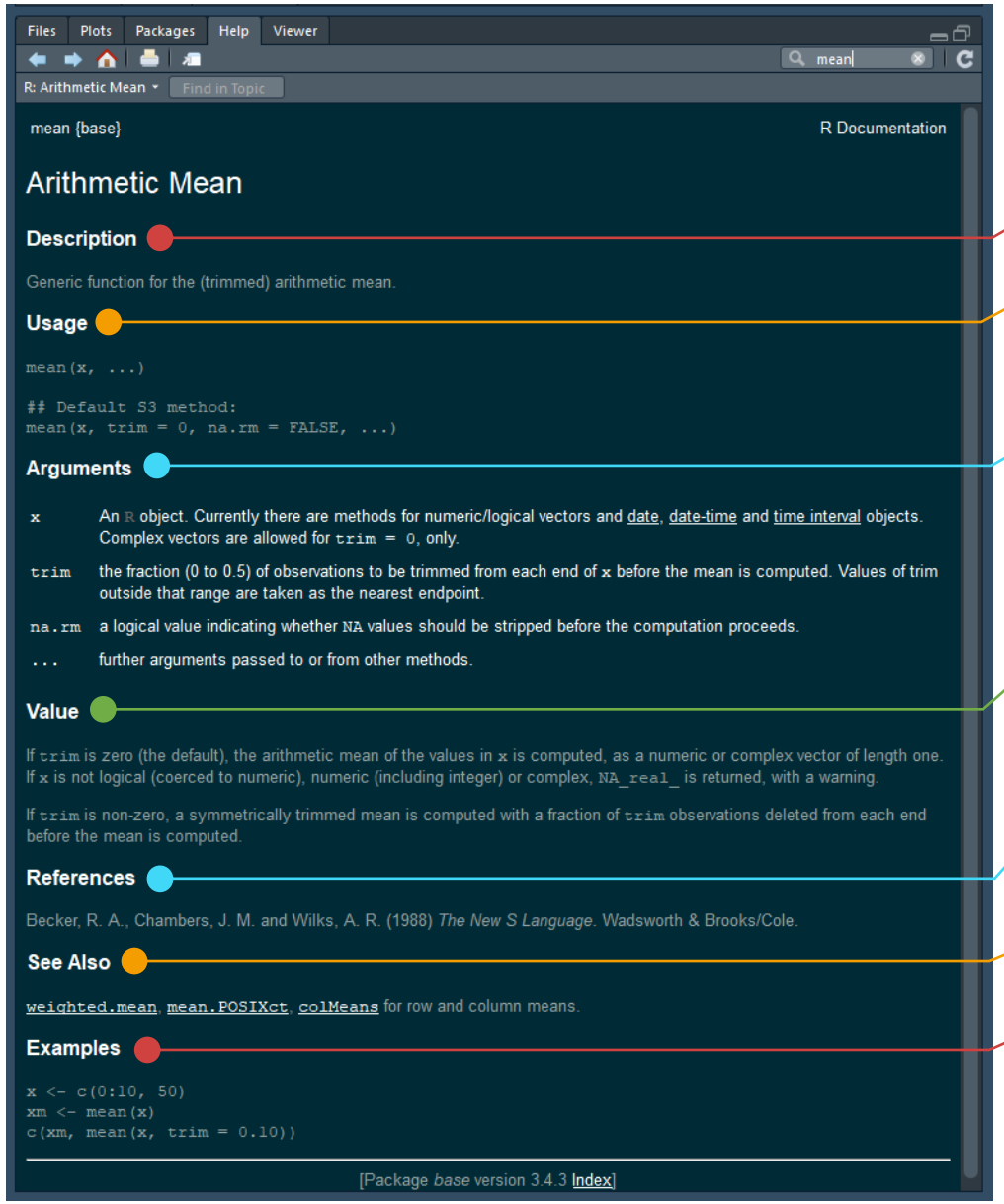
`x[2]` `x["b"]` `x[3:5]` `x[c(1,7,9)]` `x[-c(1:5,7,9)]` `x[x>13 & x<21]`

Note that indexing with a **negative** integer works best if you are keeping most of the values and only want to get rid of some. You can also subset using TRUE and FALSE values, but that isn't very useful at this stage of learning. For the curious ones, it looks like this:

```
x <- 1:5                    x[c(TRUE, FALSE, TRUE, TRUE, FALSE)]          ## 1 3 4
```

365 DataScience

Getting HELP with R


Files
Plots
Packages
Help
Viewer
mean
R: Arithmetic Mean
Find in Topic
mean {base}
R Documentation
Arithmetic Mean
Description
Generic function for the (trimmed) arithmetic mean.
Usage
mean(x, ...)
Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
Arguments
x An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for trim = 0, only.
trim the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
na.rm a logical value indicating whether NA values should be stripped before the computation proceeds.
... further arguments passed to or from other methods.
Value
If trim is zero (the default), the arithmetic mean of the values in x is computed, as a numeric or complex vector of length one. If x is not logical (coerced to numeric), numeric (including integer) or complex, NA_real_ is returned, with a warning.
If trim is non-zero, a symmetrically trimmed mean is computed with a fraction of trim observations deleted from each end before the mean is computed.
References
Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) The New S Language. Wadsworth & Brooks/Cole.
See Also
weighted.mean, mean.POSIXct, colMeans for row and column means.
Examples
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))
[Package base version 3.4.3 Index]

Description: gives you a brief overview of what a function does

Usage: an example of how the function is written; if the function has many arguments, here you will see the inherent argument order

Arguments: gives you the complete list of arguments a function can take, and the kind of information R needs for each. It also tells you what that information will be used for

Value: an explanation of what the function will return when you execute it; this is essentially what your output will be

Details/References: with more complex functions this is where the author of the function may decide to draw your attention to some specifics of the function you may want to know for highly specific uses

See also: A useful pointer to related functions

Examples: Example code of the function in action; **guaranteed to work**

Use `?function` or `help(function)`to call help on a function
Use `??keyword` to call general help on something you want to do

365 DataScience

Creating a matrix

```
matrix(seq(2, 70, 2), nrow = 5, byrow = TRUE,
       dimnames = list(c("r1", "r2", "r3", "r4", "r5"),
                       c("a", "b", "c", "d", "e", "f", "g")))
```

If `nrow =` is specified, R infers what `ncol =` should be, and vice versa

	a	b	c	d	e	f	g
r1	2	4	6	8	10	12	14
r2	16	18	20	22	24	26	28
r3	30	32	34	36	38	40	42
r4	44	46	48	50	52	54	56
r5	58	60	62	64	66	68	70

named columns

named rows

`byrow = TRUE` fills out the matrix row-by-row

- Matrices are a natural extension to vectors: while vectors are 1-dimensional collections of data, matrices are **two-dimensional arrays**
- Matrices have a **fixed number of rows and columns**
- Matrices can contain only **one basic data type**
- You can **create** a matrix in the following ways:
  - ✓ `dim(x) <- c(i, j)`, where `i` and `j` are the values for rows and columns, respectively
  - ✓ `array(data, dim = c(i, j))`
  - ✓ `matrix(data, nrow =, ncol = , byrow = , dimnames = )`
    the `byrow =` argument specifies whether the matrix ought to be filled out column-by-column or row-by-row
- **Naming the dimensions** of a matrix happens in two ways:
  - ✓ With the `rownames()` and `colnames()` functions
  - ✓ By defining the `dimnames =` argument in the `matrix()` function

Matrix operations

Just like with vectors, matrix operations happen in **element-wise** fashion

Scaling is when you do an arithmetic operation on a matrix with a single value; it happens on all the values in the matrix and effectively standardizes it

To do **arithmetic operations** with matrices, they must be of the **same size** (rows x columns)

Example: operation with same-length matrices

`m3` **<-** `m1` **+** `m2`

Adding two matrices together happens element-by-element

2	4	6	8
16	18	20	22
30	32	34	36
44	46	48	50

m1

+

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

m2

=

3	9	15	21
18	24	30	36
33	39	45	51
48	54	60	66

m3

Note that to do inner and outer matrix multiplication (linear algebra), you need to specify this to R

`m1 %*% m2`
creates the product of inner multiplication

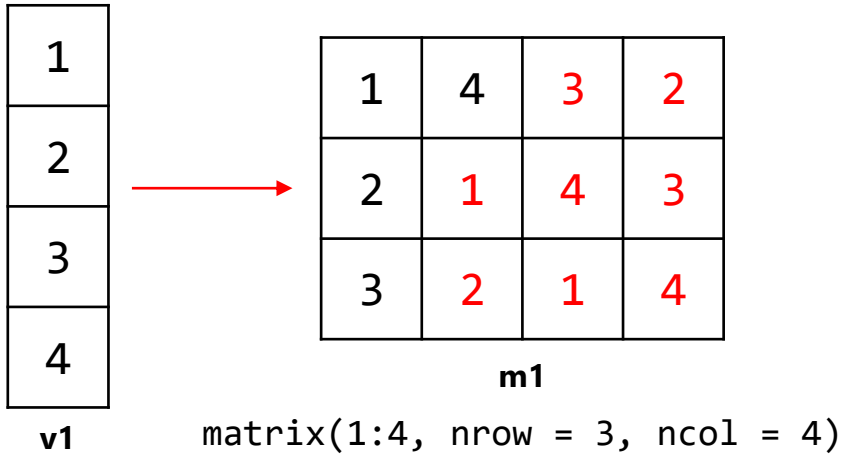
`m1 %o% m2`
creates the product of outer multiplication

Use `t()` to transpose a matrix if needed

Recycling with matrix operations

Example: creating a matrix from a shorter vector

m1 <- v1



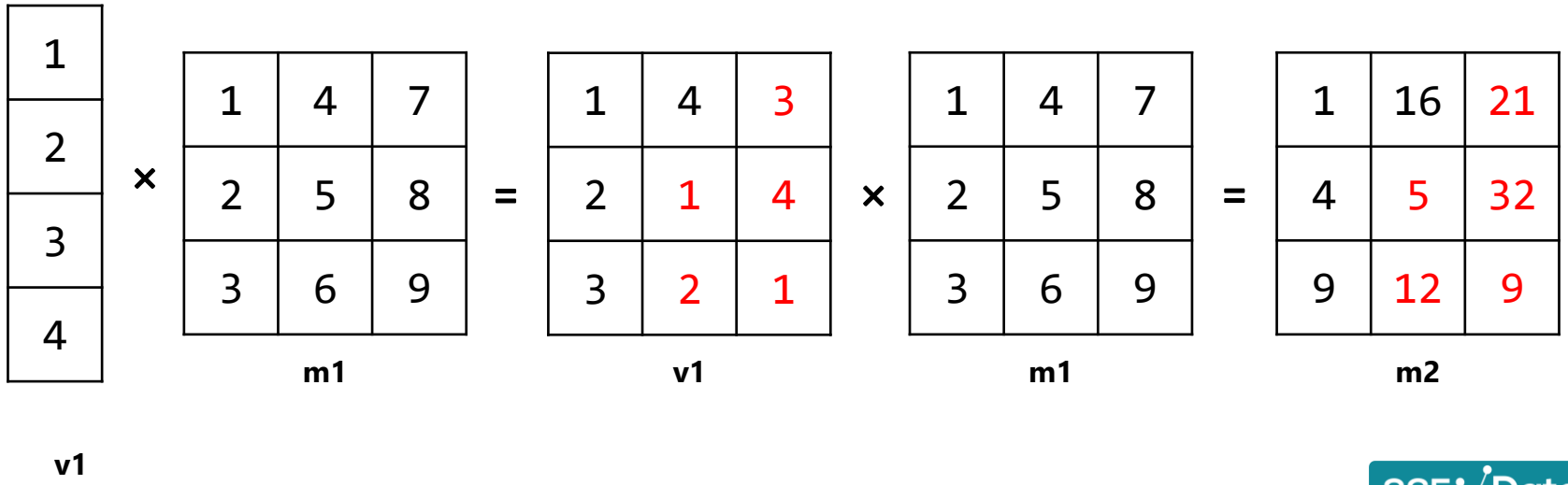

```
matrix(1:4, nrow = 3, ncol = 4)
```

Creating a matrix – you can create a matrix from a vector that does not have all the values needed to fill out the dimensions you specify, because R will recycle the vector to match the desired length

Vector × matrix operations – you can do operations with a matrix and a vector, and if the vector has fewer values than the matrix, R will again recycle

Example: multiplying a matrix with a vector

m2 <- m1 × v1



365 DataScience